Introduction to Computer Programming

A **language** is a system of communication. A **programming language** consists of all the symbols, characters, and usage rules that permit people to communicate with computers. Some programming languages are created to serve a special purpose (e.g., controlling a robot), while others are more flexible general-purpose tools that are suitable for many types of applications. However, every programming language must accept certain types of written instructions that will enable a computer system to perform a number of familiar operations. That is, every language must have **instructions** that fall into the following familiar **categories**:

1. Input/output instructions

Required to permit communication between I/O devices and the central processor, these instruction provide details on the type of input or output operation to be performed and the storage locations to be used during the operation.

2. Calculation Instruction

Instruction to permit addition, subtraction, multiplication, and division during processing are of course common in all programming languages.

3. Logic/Comparison Instructions

These instructions are used to transfer program control, and are needed in the selection and loop structures that are followed to prepare programs. During processing, two data items may be compared as a result of the execution of a logic instruction. As you know, program control can follow different paths depending on the outcome of a selection test (**IF R > 0, THEN A, ELSE B**). And a **loop** can be continued or terminated depending on the outcome of an exit condition test (does Q=-99.9?). In addition to the instruction in languages that the set up tests or comparison to effect the transfer of program control, there are also unconditional transfer instructions available that are not based on the outcome of comparisons.

4. Storage/retrieval and Movement Instructions

These instructions are used to store, retrieve, and move data during processing. Data may be copied from one storage location to another and retrieve as needed.

But even though all programming languages have an instruction set that permits these operations to be performed, there's a marked difference to be found in the symbols, characters, and syntax of machine languages, assembly languages, and high-level languages.



Programming Language Classification

Machine Languages

A computer's **machine language** consists of strings of binary numbers and is the only one the CPU directly "understands." An instruction prepared in any machine language will have at least two parts. The first part is the command or operation, and it tells the computer what function to perform. Every computer has an **operation code** or "op code" for each of its functions. The second part of the instruction is the **operand**, and it tells the computer where to find or store the data or other instructions that are to be manipulated. The number of operands in an instruction varies among computers. In a single-operand machine, the binary equivalent of "ADD 0184" could cause the value in address 0184 to be added to the value stored in a register in the arithmetic-logic unit. In a two operand machine, the binary representation for "ADD 0184 8672" could cause the value in address 8672 to be added to the number in location 0184. the single operand and format is popular in the smallest microcomputers; the two operand structure is likely to be available in most other machine.

By today's standards, early computers were intolerant. Programmers had to translate instructions directly into the machine-language form that computers understood. For example, the programmer writing the instruction to "ADD o184" for an early IBM machine would have written:

00010000000000000000000000001011000

Assembly Language

To ease the programmer’s burden, mnemonic operation codes and symbolic addresses were developed in the early 1950s. The word mnemonic (pronounced ne-monik) refers to a memory aid. One of the first steps in improving the program operation process was to substitute letter symbols for the numeric machine-language operation codes. Each computer now has a **mnemonic code**, although, of course, the actual symbols vary among makes and models. The following figure shows mnemonic codes for few commands used with some IBM mainframe computer:

Command name	Mnemonic Code	Command name	Mnemonic Code
Input /Output Commands		Compare Logical Character	CLS
Start I/O	SIO	Branch on Conditional Registrar	BCR
Halt I/O	HIO	Branch on Condition	BC
Calculation Commands		Branch on count	BCT
Add	A	**Storage/Retrieval &Movement**	
Subtract	S	Load Registrar	LR
Multiply	M	Load	L
Divide	D	Move Character	MVC
Logic/Comparison Commands		Move Numeric	MVN
Compare Registrar	CR	Store	ST
Compare	C	Store Character	STC

Machine language is still used by the computer as it processes data, but **assembly language** software first translates the specified operation code symbol into its machine-language equivalent.

And this improvement set the stage for further advances. If the computer could translate convenient symbols into basic operations, why couldn't it also perform other clerical coding functions such as assigning storage addresses to data? **Symbolic addressing** is the price of expressing an address not in items of its absolute numerical calculation, but rather in terms of symbols convenient to the programmer.

In the early stages of symbolic addressing, the programmer assigned a symbolic name and an actual address to a data item. For example, the total value of merchandise purchased during a month by a department store customer might be assigned to address 0063 by the programmer and given the symbolic name **TOTAL**. The value of merchandise return unused during the month might be assigned to address 2047 and given the symbolic name **CREDIT**. Then, for the remainder of the program, the programmer would refer to the symbolic names rather than to the address when such items were to be processed. Thus, an instruction might be written "**S CREDIT, TOTAL**" to subtract the value of returned goods from the total amount purchased to find the amount of the customer's monthly bill. The assembly language software might then translate the symbolic instruction into this machine-language string of bits:

011111	**011111111111**	**000000111111**
Mnemonic op code	**2047**	**0063**
(S)	**(CREDIT)**	**(TOTAL)**

Another improvement followed. The programmer turned the task of assigning and keeping track of instruction addresses over to computer. The programmer merely told the machine the storage address number of the first program instruction, and assembly language software then automatically stored all other in sequence from that point. So if another instruction was added to the program later, it was not necessary to modify the addresses of all instructions that followed the point of instruction. Instead, the processor automatically adjusted storage locations the next time the program ran.

This assembly program, or assembler, also enables the computer to convert the programmer's assembly language instruction into its own machine code. A program of instructions written by programmer in an assembly language is called a **source program**. After this source program has been converted into machine code by an assembler, it's referred to as an **object program**. It's easier for programmers to write instructions in an assembly language than to prepare instructions in machine-language code. But two computer runs may be required before source program instruction can be used to produce the desired output.

Assembly languages have advantages over machine languages. They save time and reduce detail. Fewer errors are made, and those that are made easier to find.

And assembly programs are easier for people to modify than machine language programs. But there are limitations. Coding in assembly language is still time consuming. And a big drawback of assembly languages is that they are machine oriented. That is, they are designed for specific make and model of processor being used. Programs might have to be recorded for different machine.

High-Level Languages

The earlier assembly programs produced only one machine instruction for each source program instruction. To speed up coding, assembly programs were developed that could produce a variable amount of machine language code for each source program instruction. In other words, a single **macro instruction** might produce several lines of machine-language code. For example, the programmer might write **"READ FILE"** and the translating software might then automatically provide a detailed series of previously prepared machine-language instructions which would copy a record into primary storage from the file of data being read by the input device. Thus, the programmer was relieved of the task of writing an instruction for every machine operation performed.

The development of mnemonic techniques and macro instructions led, in turn, to the development of **high-level languages** that are often oriented toward a particular class of processing problems. For example, a number of languages have been designed to process problems of scientific-mathematic nature, and other languages have appeared that emphasize file processing applications.

Unlike assembly, high-level language programs may be used with different makes of computers with modification. Thus, reprogramming expense may be greatly reduced when new equipment is acquired. Other advantages of high-level language programs are:

- They are easier to learn than assembly languages.
- They are require less time to write.
- They provide better documentation.
- They are easier to maintain.
- A programmer skilled in writing programs in such a language is not restricted to using a single type of machine.

Compiler Translation. Naturally, a source program written in high-level language must also be translated into machine-usable code. A translating program that can perform this operation is called a **compiler**. Compilers, like advanced assembly programs, may generate many lines of machine code for each source program statement. A compiling run is required before problem data can be processed. With the exception that a compiler program is substituted for an assembly program, the procedures are essentially the same as assembly language source program instructions. The production run follows the compiling run.

Interpreter Translation. An alternative to using a compiler for high-level language translation is often employed with personal computers. Instead of translating the source program and permanently saving object code produced during a compiling run for future production use, the programmer merely loads the source program into the computer along with idea to be processed. A permanently hardwired interpreter program located inside the computer then converts each source program statement into machine-language form as it's needed during the processing of the data. No object code is saved for future use.

Major High-Level Languages used in Program Coding

High level programming Languages can be **categorized** into two main categories:

1- Object Oriented Programming languages (**OOP**), which are newly in use.
2- Procedural Programming Languages (**PPL**), which are the old traditional ones.

Object-oriented programming (OOP) is a programming language model organized around "objects" rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data. The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them.
One of the first object-oriented computer languages was called ***Smalltalk;*** *other latter languages such as* ***C++, C#, Java, python, Visual FoxPro,*** *and* ***Visual Basic*** *(VB)* ***(also VB.net)*** are the most popular object-oriented languages today. The Java programming language is designed especially for use in distributed applications on corporate networks and the Internet. (in this stage we will not expand in these languages except ***VB***).

In **Mechatronics**, there are many **Robot** and **control** programming languages, but there is no standard language yet, each manufacturer company has its own language to use.

Procedural Programming (PPL) is the traditional well known languages.
Early work on high-level languages began in the 1950s. **Dr. Grace M. Hopper**, for example, developed a compiler (named A-2) in 1952. Since then, many other high-level languages have been produced. Let us take a look now at a few of the most popular high-level languages:

FORTRAN, **COBOL**, **Basic**, **Pascal**, **PL/1, C**, **Logo**…etc.

